



Sorting on GPUs

Revisiting some algorithms from lecture 6:

Some not-so-good sorting approaches

Bitonic sort

QuickSort

Concurrent kernels and recursion



Adapt to parallel algorithms

Many sorting algorithms are highly sequential

Suitable for parallel implementation?

- **Data driven execution**
- **Data independent execution**



Data driven execution

Computing pattern depends on data

Usually harder to parallelize!

Example: QuickSort.



Data independent execution

Known computing pattern

Easier to parallelize - always the same plan

Example: Bitonic sort



Bubble sort

Loop through data, compare neighbors

Extremely sequential

Inefficient

Parallel version: Bubble sort with odd-even transposition method

Compare all items pairwise

Two phases, "odd phase" and "even phase" (shifted one step)



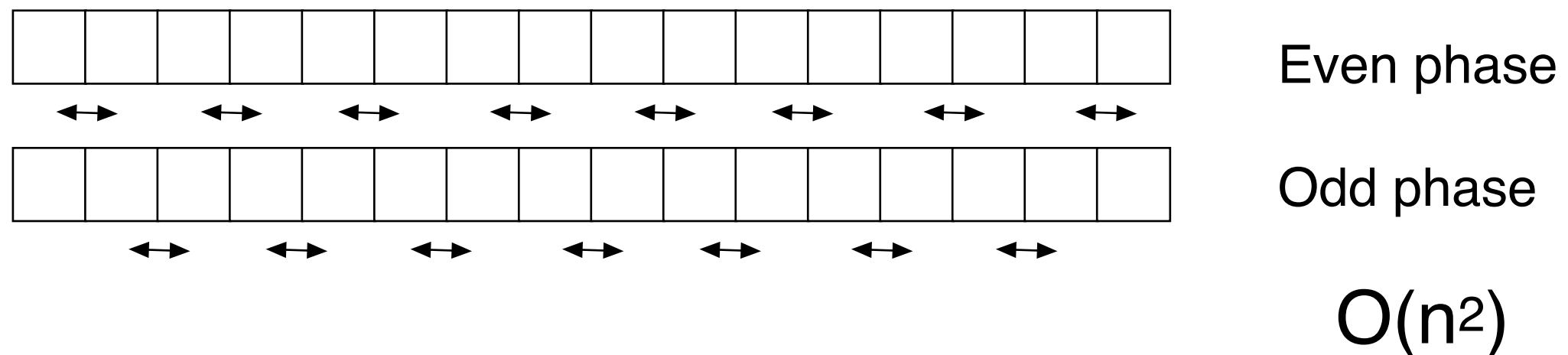
Bubble sort, parallel version

Bubble sort with odd-even transposition method

Compare all items pairwise

Two phases, "odd phase" and "even phase" (shifted one step)

Fully sorted after n phases





Suitable for GPU?

Not as bad as it seems at first look:

- **Data independent**
 - **Excellent locality**
 - **Pretty good possibilities to use shared memory (but with some costly transfers at edges between blocks). Thus close to optimal in global memory transfers.**
 - **But certainly not optimal at very large sizes**
- ”Better” algorithms don’t necessary beat this all that easily!**



Rank sort

Count number of items that are smaller

Easy to parallelize:

- **One thread per item**
- **Loop through entire data**
- **Store in index decided from count of number of smaller items.**



Suitable for GPU?

Again, not as bad as it seems at first look:

- **Data independent**
- **Excellent locality - especially good for broadcasting (e.g. constant memory). Also suitable for shared memory.**
 - **Again, $O(n^2)$: Will grow at very large sizes**

Two bad ones that are not quite as bad as they seem.

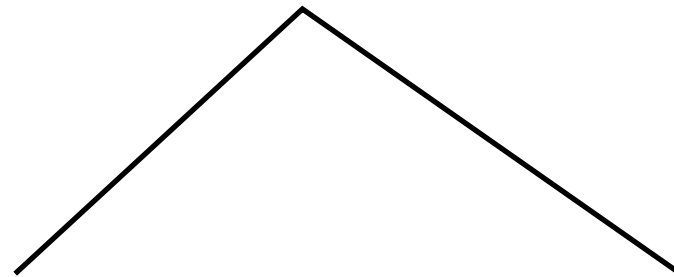
N parallel iterations may beat $N \log N$ sequential ones!



Bitonic sort

(As described in lecture 6)

Bitonic set: Two monotonic parts in different direction.





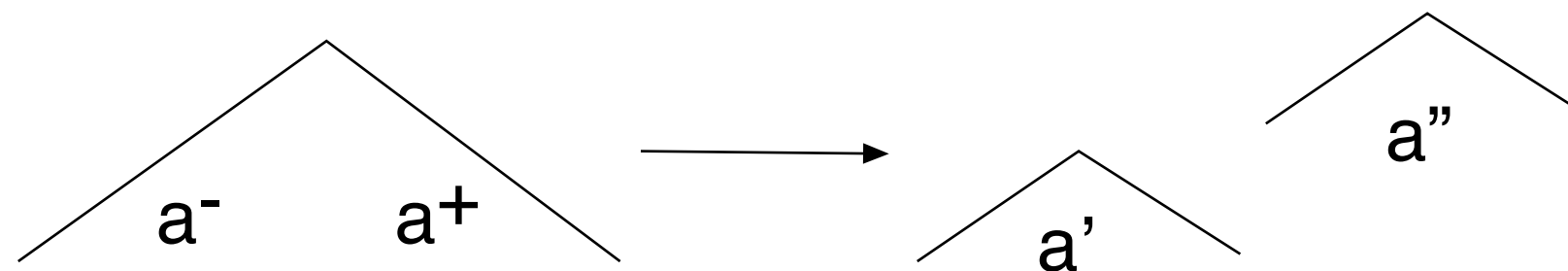
Bitonic sort

(According to Batcher:) Let a be a bitonic set with a maximum at k , consisting of two monotonic parts, one increasing, a^- (from item 1 to k) and one decreasing, a^+ ($k+1$ to n)

Then two new sets can be constructed as

$$a' = \min(a_1, a_{k+1}), \min(a_2, a_{k+2}) \dots$$
$$a'' = \max(a_1, a_{k+1}), \max(a_2, a_{k+2}) \dots$$

These two sets are also bitonic and $\max(a') \leq \min(a'')$!





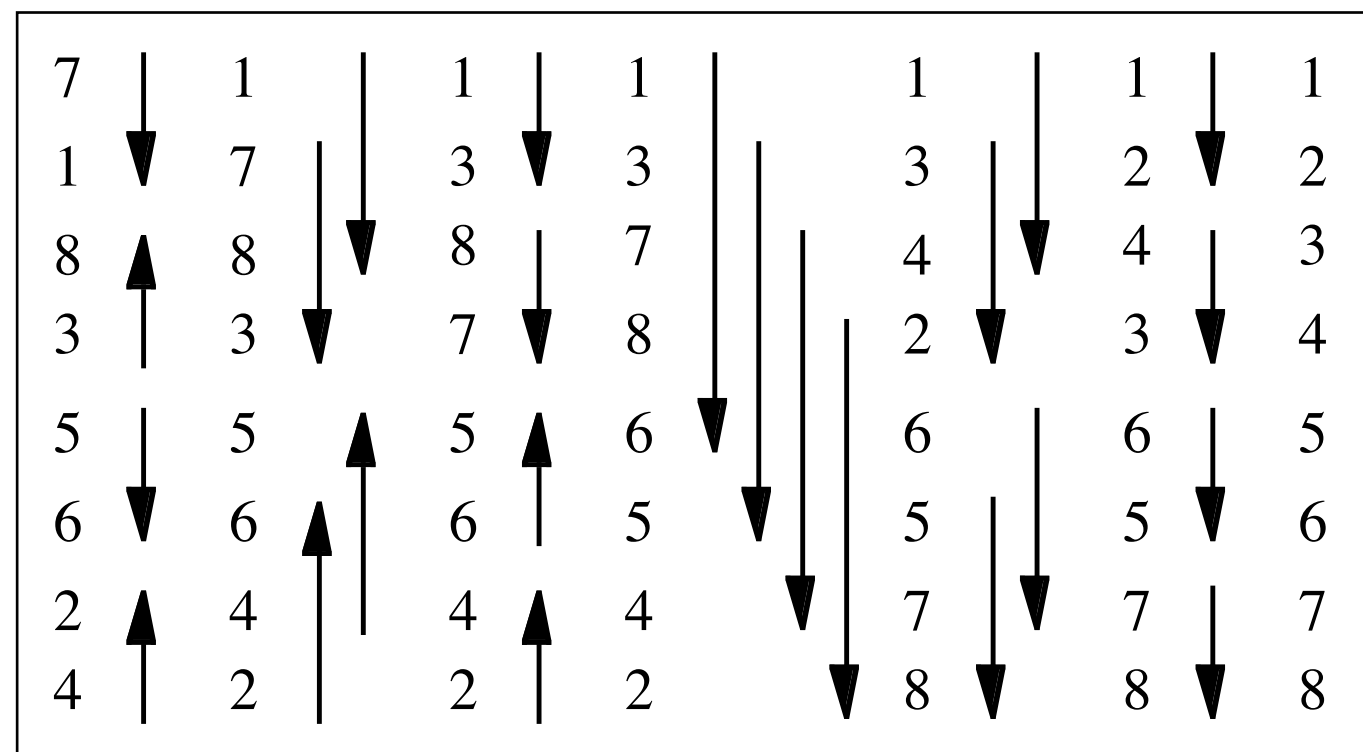
Bitonic sort by divide-and-conquer

**Bitonic sort works on a bitonic sequence:
partially sorted**

**The parts must be sorted. Sort them by
bitonic sort!**



Bitonic sort example



Bitonic sort of
smaller parts

Bitonic sort of main
part

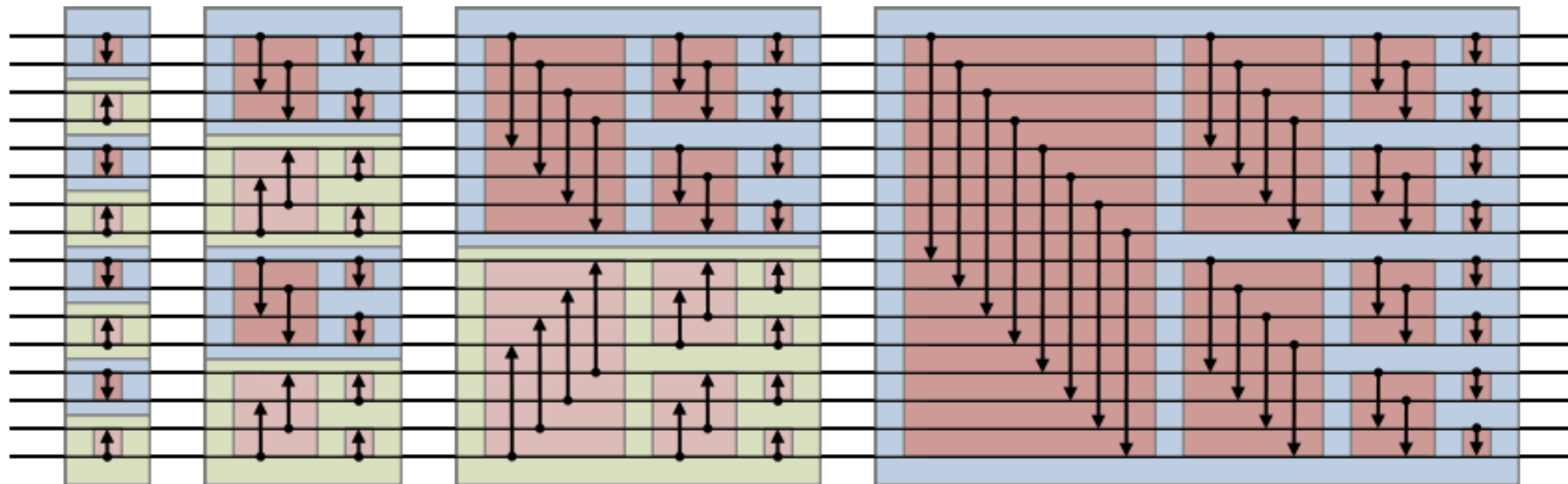
Reverse parts
(bitonic merge)

Reverse parts
(bitonic merge)



Bigger example

The problem scales nicely, uniformly



More stages gives longer stages

(Image from Wikipedia)



Get those steps right

Step length

Step direction

Comparison direction

Calculated from stage number and stage length



Code examples

Sequential

Recursive example

Iterative example



Bitonic sort

- **Data independent, no worst case**
 - **Fast: $O(n \cdot \log^2 n)$ (Why?)**
 - **Good locality in some parts**
- but**
- **Big leaps in addressing for some parts**



What about those big leaps?

**Small leaps: Can be computed within one block.
Shared memory friendly.**

**Big leaps (>number of threads/block): No
synchronization possible between blocks!**

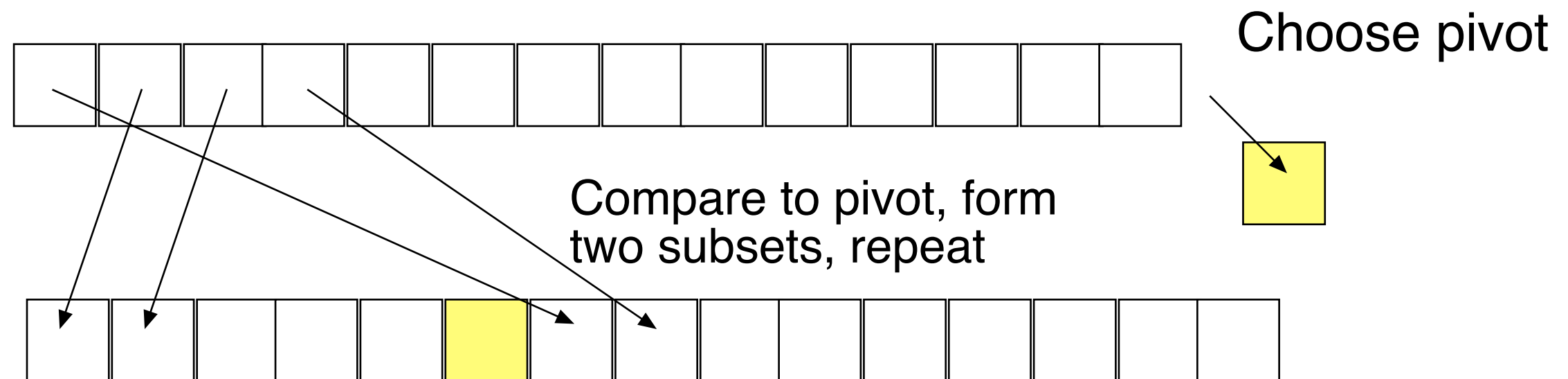
But we *must* synchronize!

-> multiple kernel runs!



QuickSort

Very popular algorithm for sequential implementation



Data driven, data dependent reorganization, non-uniform

Fancy name - nobody expect QuickSort to be nothing but optimal



QuickSort is

Fast: $O(n \cdot \log n)$ in typical cases

$O(n^2)$ in the worst case

Data driven, data dependent reorganization, non-uniform

**Fancy name - nobody expects QuickSort to be nothing
but optimal**



QuickSort on GPU

Initially ignored as impractical

CUDA implementations exist

**Data driven approaches increasingly suitable as
GPUs become more flexible**



Parallel QuickSort

Several stages to consider:

- **Pivot selection. Usually just grab one.**
 - **Comparisons**
 - **Partitioning**
- **Concatenate result**



Pivot selection

If we could always pick a pivot that splits the data in half...





but you can't do that without sorting! But how about a random one?



There is a worst case caused by bad pivots. Live with it!



Comparisons

Easy to parallelize

**One thread per comparison not unreasonable!
(GPUs don't have a problem with many threads!)**

No problem!



Partitioning

The big problem!

Sequential partitioning: Bad!

**Parallel partitioning 1: Atomic fetch & increment.
(GPUs have atomics!)**

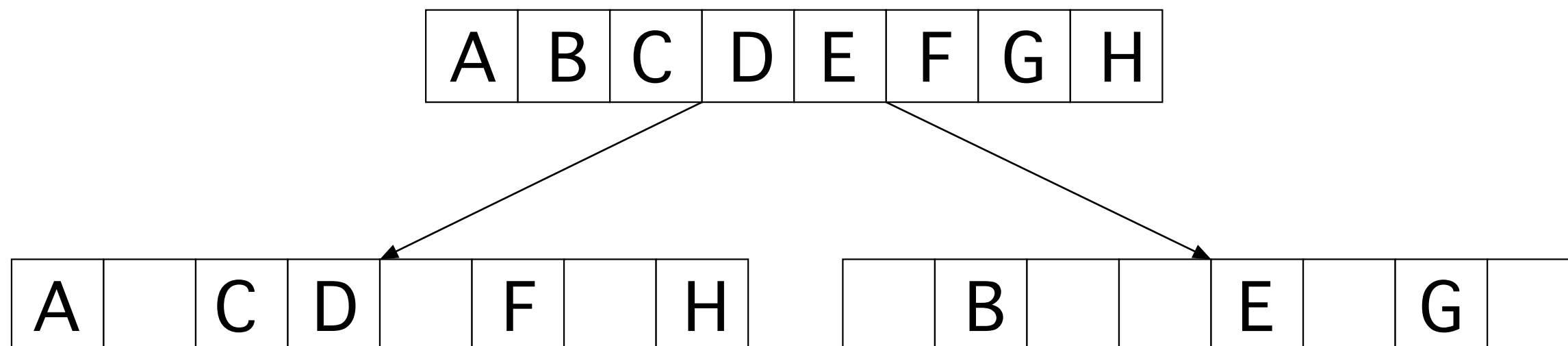
Parallel partitioning 2: Divide and conquer



In-place sorting not feasible

Split to two list of same size as original. Massive number of threads!

Then we must pack to smaller size.





Packing to smaller size not trivial

Data dependent

One possibility: Count occupied spaces to simplify the problem. Create a look-up table for addressing.

A		C	D		F		H
1	0	1	1	0	1	0	1
1		2		1		1	
3				2			
5							



Thus, QuickSort is not impossible, but more complex than before.

Note:

GPUs have Compare-And-Swap atomics!

GPUs favor massive numbers of threads. One thread per comparison is more than OK!

See also lecture 5 and 6



Recursion

GPUs can't do recursion efficiently... or can they?

Since Kepler we have *concurrent kernels*

Not only a matter of launching kernels from CPU!

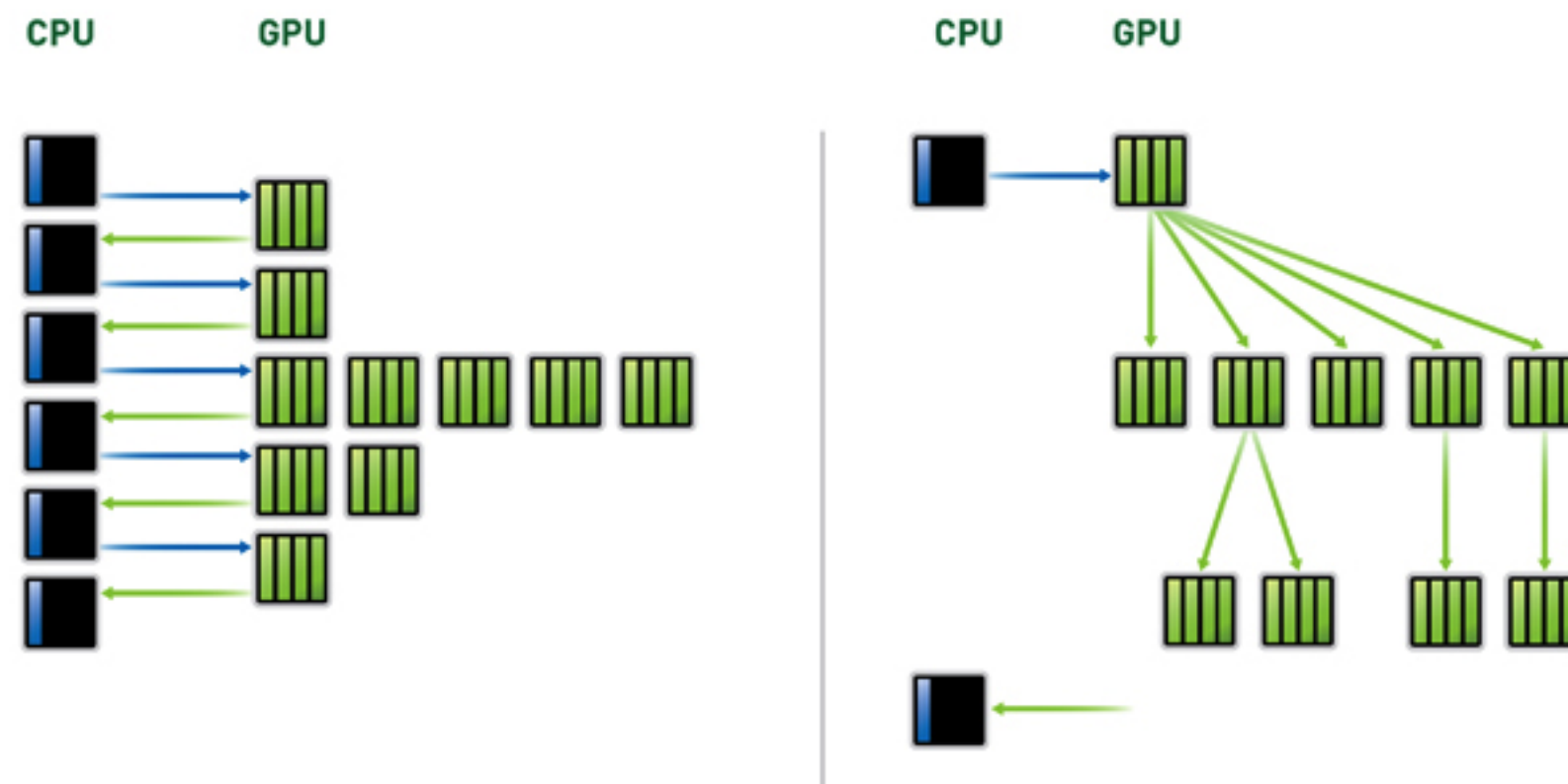
A kernel can spawn new kernels!

Do recursion by spawning new kernels!



Concurrent kernels, Dynamic Parallelism

Less work for the CPU to manage the computation.





Recursion can look like this:

```
__global__ void quicksort(int *data, int left, int right)
{
    int nleft, nright;
    cudaStream_t s1, s2;

    // Partitions data based on pivot of first element.
    // Returns counts in nleft & nright
    partition(data+left, data+right, data[left], nleft, nright);

    // If a sub-array needs sorting, launch a new grid for it.
    // Note use of streams to get concurrency between sub-sorts
    if(left < nright) {
        cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);
        quicksort<<< ..., s1 >>>(data, left, nright);
    }
    if(nleft < right) {
        cudaStreamCreateWithFlags(&s2, cudaStreamNonBlocking);
        quicksort<<< ..., s2 >>>(data, nleft, right);
    }
}

__host__ void launch_quicksort(int *data, int count)
{
    quicksort<<< ... >>>(data, 0, count-1);
}
```

But... does this really
do a good job on
partitioning?

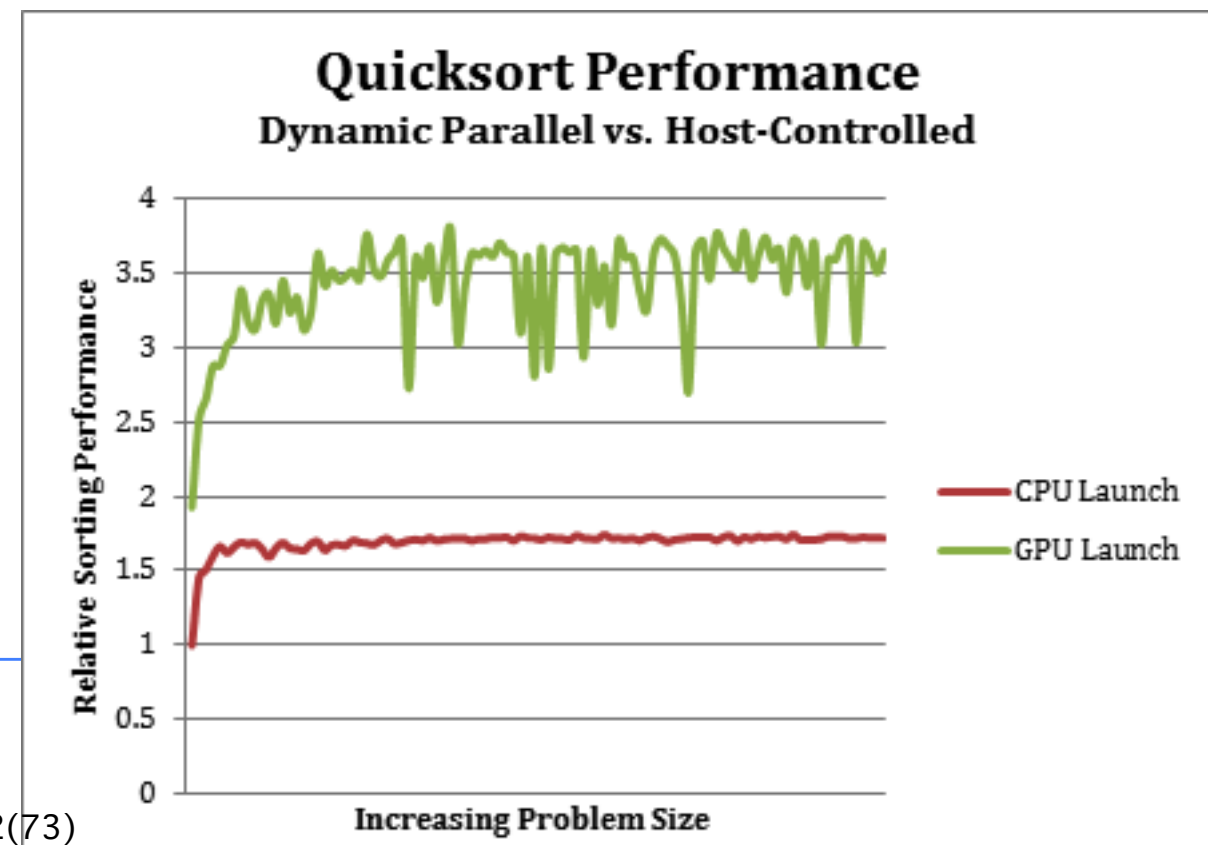
Source: <http://blogs.nvidia.com/blog/2012/09/12/how-tesla-k20-speeds-up-quicksort-a-familiar-comp-sci-code/>



Advantages

- **Less work for CPU**
- **Less synchronizing (from CPU side)**
- **Easier programming!**

They claim it matters this much (but your milage will vary)





Information Coding / Computer Graphics, ISY, LiTH

Recursive CUDA kernels, a promising improvement

Big change in GPU computing?

Southfork has GPUs that support it.